CHAPTER

SEVEN

EXCHANGE FUNCTIONS

Exchange Functions is a specification mechanism for designing and a model for describing distributed and embedded systems.* Exchange Functions assumes explicit processes that communicate by calling functions that exchange values. Communication with exchange functions is bidirectional, simultaneous, and symmetric. That is, in an exchange (communication), information transfers between both communicators simultaneously; each communicator has an equal role in establishing the communication. The model has mechanisms for both blocking and nonblocking communication. Exchange functions are a particularly elegant integration of communication and communication failure. Pamela Zave and D. R. Fitzwater developed the Exchange Functions model [Fitzwater 77].

Varieties of Exchange Functions

Exchange Functions is based on synchronous, simultaneous, and bidirectional communication on a static, predetermined set of processes. Each process is a state automaton, described by a current state and a function for reaching its next state. With Turing machines, state progression is determined by the machine's internal state (tape), state-transition function, and input. Similarly, state transition in Exchange Functions is defined by the process's state, state transition function (its *successor function*), and the value received in a communication exchange. For

* An *embedded system* is a computer system that is incorporated in a larger device. For example, a computer used to monitor and control a manufacturing process is an embedded system.

example, the even process (which does no communication) has as its successor function:

$$\text{successor(even)} \equiv \text{even} + 2$$

If this process's initial state is 0, it successively takes as its states the even numbers: 0, 2, 4, ...

Successor functions are built up by functional composition from other, more primitive functions. Besides the usual kinds of primitives (such as arithmetic, conditional, constructor, and selector primitives), Exchange Functions introduces a new set of communication primitives—the exchange functions. Two processes that call corresponding exchange functions communicate. Each exchange function takes a single argument (the value to be sent to another process) and returns a single value (the value received from the other process). Exchange functions thus unify input and output.

Each exchange function refers to a particular channel. Two exchange functions can communicate only if their channels match. Calls to exchange functions on different channels do not result in communication. We let E_α indicate a call of exchange function E on channel α. If process P executes an $\mathsf{E}_\alpha(1)$ and process Q executes an $\mathsf{E}_\beta(2)$, no exchange occurs (as α and β are different channels). If Process R then executes an $\mathsf{E}_\beta(3)$, then a 3 is returned as the value of Q's call, and a 2 as the value of R's call. P remains blocked, waiting for a call on channel α.

The model defines three varieties of exchange functions, distinguished by their matching and temporal characteristics. The simplest is X. Evaluating X (on some channel) causes the process to block until another process evaluates any other exchange function on the same channel. When this happens the two processes exchange arguments and return. Thus, X is a waiting, synchronous primitive.

Calling X blocks a process until another process executes a matching exchange function. Sometimes a process needs to check whether another process is trying to communicate, but wants to avoid blocking if no communication is available. Primitive function XR serves this purpose. If a process executes an XR and some other process is currently waiting on the same channel, they exchange and return. If no process is waiting on that channel, the argument of the XR is returned. A process issuing an XR distinguishes between successful and unsuccessful communication attempts by calling XR with an argument that could not be the result of a successful exchange. That is, if P executes $\mathsf{XR}_\alpha(4)$, and no communication is waiting on channel α, then the value of $\mathsf{XR}_\alpha(4)$ is 4. The R in XR stands for real time. Zave and Fitzwater assert that this exchange behavior is essential for real-time systems.

Metaphorically, we compare the XR primitive to a "flashing" liquid crystal display (LCD) clock. Each second, the clock offers the time. A person looking at the clock (waiting for the time) sees the time when the clock flashes. The clock flash is instantaneous. The clock never waits for the time seeker; the time seeker always waits for the clock. This parallels real-time sensing, where the sensor misses events if it is not watching for them.

Any X_α can communicate with any other X_α. This uniform matching makes it awkward to program some algorithms that use a many-to-one communication pattern. For example, many processes may share a single buffer. They need to communicate with the buffer, not each other. The XM exchange function provides the needed "directionality" to communication. Briefly, calls to XM do not exchange with other calls to XM. XM calls can still exchange with Xs and XRs on the same channel. The M in XM stands for many-to-one.

One example of the use of XM is Zave's description of a real-time clock [Zave 82]. The clock is a process that executes $\mathsf{XR}_{\text{clock}}$(current_time) each "tick." This offers the clock's time to any process that desires it. To get the time, processes execute $\mathsf{XM}_{\text{clock}}$(time_please) (where time_please is any arbitrary value.) This call waits for a matching $\mathsf{XR}_{\text{clock}}$. If instead of XM, time-desiring processes execute $\mathsf{X}_{\text{clock}}$(time_please), then two processes requesting the time could communicate with each other (sending each other a time_please), instead of receiving the time from the clock. If the clock updates the value of current_time between calls to $\mathsf{XR}_{\text{clock}}$, then no two processes ever receive the same time.

The check marks in Table 7-1 show the possible communication matches between the three exchange primitives. The variety of exchange function primitives (X, XR, and XM) selects all useful possibilities in a two-by-two grid. An exchange function can exchange with itself, like X (self-exchange) or exchange only with other functions, like XR and XM. An exchange function can be blocking (waiting) like X and XM or it can be instantaneous (nonwaiting), like XR. An instantaneous primitive is available only for an instant. Therefore, an instantaneous primitive never exchanges with another instantaneous primitive. Table 7-2 illustrates this relationship.

Zave and Fitzwater assert that the channel of any particular call to an exchange function must be a compile-time constant [Zave 77]. This has two ramifications. The first is that one cannot subscript channel names in an Exchange Functions program. The second is that channels cannot be dynamically created

Table 7-1 Potential exchanges

	X	XR	XM
X	√	√	√
XR	√		√
XM	√	√	

Table 7-2 Exchange function dimensions

	Waiting	Instantaneous
Self-exchange	X	*Impossible*
No self-exchange	XM	XR

and transferred between processes. In practice, this restriction is just a syntactic impediment—one can achieve subscripting on channel names (over a known set of channels) by a sufficiently complicated program structure. We believe that it is a mistake not to include subscripted channels in the model. So we ignore this restriction in our examples.

Implementing Exchange Functions in a distributed network requires *conflict resolution*—the matching of interacting pairs. Zave and Fitzwater specify that this conflict resolution be weakly fair—that is, no pending exchange (from an X or XM) should be indefinitely denied [Zave 77].

Zave suggests that when more than two processes communicate on the same channel, the communication pattern is almost always many-to-one (though the particular application determines which of the pairs XM/XR or XM/X is appropriate [Zave 83]). Later in this section we present a fanciful counterexample to this hypothesis, a program with a completely unfocused communication pattern.

Sometimes one has a choice of several possible exchange patterns for a particular application. For example, an X–X communication between two processes can just as easily be performed with an X–XM pair; the effect of an X can sometimes be achieved by performing an XR in a loop. (However, one does not achieve an X–X communication pattern using two looping XRs!)

The communication mechanisms of Exchange Functions parallel, to some extent, the communication facilities provided by shared-loop bus systems such as the Ethernet [Metcalfe 76]. The shared buses of Ethernet correspond to the channels of Exchange Functions.

Successor Functions

In Exchange Functions, processes have state. The state of a process changes, stepwise, through the life of the process. Each process has a successor function that describes this change. This function, given the state of a process, returns the new state of the process. The evaluation of this function may block, pending completion of its communications. Notationally, we indicate the successor function for a process by applying the successor function to the name of that process.

The process identifier names the state of the process. For example, if F is the successor function of process P, we write

$$\mathsf{successor(P)} \equiv \mathsf{F(P)}$$

Viewing the process identifier as a variable, this sequence of states is analogous to the program

$$\textbf{while true do } \mathsf{P := F(P)}$$

Following Zave [Zave 82], we use functional notation to describe successor functions.*

Binary semaphore A binary semaphore is a process with communication capabilities on two channels, Psem and Vsem. When the semaphore is free, it executes $\mathsf{X}_{\mathrm{Psem}}$. When the call on Psem returns, the semaphore calls $\mathsf{X}_{\mathrm{Vsem}}$. The semaphore alternates calls to $\mathsf{X}_{\mathrm{Psem}}$ and $\mathsf{X}_{\mathrm{Vsem}}$. The state of the semaphore is expressed entirely by the channel on which it is waiting. A simple program for a binary semaphore is

$$\mathsf{successor\ (semaphore)} \equiv \mathsf{X}_{\mathrm{Vsem}}(\mathsf{X}_{\mathrm{Psem}}(\odot))$$

Shared Variables The Lynch-Fischer model (Chapter 6) is based on shared variables. In that model, the only communication mechanism between processes is the reading and writing of shared variables. The processes of the general Lynch-Fischer model are powerful automata. In a single atomic step they can read a shared variable, compute a new value for that variable, and write that value back into the variable.

We model shared variable communication in Exchange Functions by creating a register process. For each register we have two channels, read, to be used in reading the register value, and write, to be used in writing it. Like the semaphore, the register forces an order on these operations: the alternations of reads and writes. Unlike the semaphore, the register retains its state between reads and writes. The program for the register is

$$\mathsf{successor(register)} \equiv \mathsf{X}_{\mathrm{write}}\ (\mathsf{X}_{\mathrm{read}}\ \mathsf{(register)})$$

* Briefly, the notation $\mathsf{F\,(G\,(H\,(x), y))}$ is equivalent to the sequential program

```
function FGH (x);
begin
    tempH  := H(x);
    tempG  := G(tempH, y);
    answer := F(tempG);
    return (answer)
end
```

Operationally, the register responds to a call on X_{read} by sending the current value of the register. The value returned (a synchronization signal like $\odot$) is used as the argument to $\mathsf{X}_{\text{write}}$; the value returned by the call on write becomes the new value of the register. Thus, the call on read sends the register's value and receives a synchronization signal; the call on write sends a synchronization signal and receives the register's new value.

Processes that access the register include the sequence

$$\ldots\ \mathsf{XM}_{\text{write}}\ (\mathsf{F}\ (\mathsf{XM}_{\text{read}}\ (\odot)))\ \ldots$$

as part of their successor function. To be true to the pure Lynch-Fischer model, function F should not read or write any other shared variables.

Unbounded buffer Our unbounded producer-consumer buffer executes the following algorithm: At each step the buffer calls XR on the producer channel and XR on the consumer channel. The buffer changes its state if either call is matched. If neither channel has a waiting call, the buffer retains the same state. Let

Buffer	The ordered list that represents the state of the buffer process.
nil	The null (empty) buffer.
Ack-from-C	The acknowledgment from the consumer.
Ack-to-P	The acknowledgment to the producer.

We represent the state of a buffer as a list L of elements L_1, L_2, ..., L_k (where k is the length of L). We then define the following functions (with their Lisp equivalents, of course, in parentheses):

```
first (L)            ≡  L_1                                  -- (car L)
rest (L)             ≡  L_2 ... L_k                          -- (cdr L)
first-insert (e, L)  ≡  if (e = Ack-from-C) then L
                        else e L_1 L_2 ... L_k               -- (cons e L)
last-insert (L, e)   ≡  if (e = Ack-to-P) then L
                        else L_1 L_2 ... L_k e               -- (append L (list e))
```

The successor function of the buffer is

```
successor (Buffer) ≡
    if Buffer = nil then last-insert (Buffer, X_p(Ack-to-P))              -- (*)
    else last-insert (first-insert (XR_c (first (Buffer)),
                                    rest (Buffer)),
                      XR_p(Ack-to-P))
```

This buffer provides two communication channels: XR_{c} for communication with consumers and XR_{p} for communication with producers. An empty buffer is receptive only to messages from producers. The then clause of the conditional handles this possibility (*). Steps of the buffer process fit into one of four different patterns: no messages sent to the buffer, messages only from producers, messages only from consumers, and messages from both producers and consumers. If no messages are sent on either channel, then the $\mathsf{XR_c}$ is first (Buffer) and the value of $\mathsf{XR_p}$ is Ack-to-P. The computation proceeds as

```
successor(Buffer)
    ⇒ last-insert (first-insert (first(Buffer), rest (Buffer)), Ack-to-P)
    ⇒ first-insert (first (Buffer), rest (Buffer))
                          -- as last-insert(B, Ack-to-P) = B
    ⇒ Buffer
```

If a process tries to consume, $\mathsf{XM_c}$(Ack-from-C) and $\mathsf{XR_c}$(first (Buffer)) exchange. Ack-from-C is ignored by first-insert (when Buffer is nil) and the buffer shrinks. If a process tries to produce, $\mathsf{XM_p}$(value) exchanges with an $\mathsf{XR_p}$(Ack-to-P), this last-insert (Buffer, value) successfully adds a new element at the end of the buffer. These two exchanges can occur on the same successor step; the buffer shrinks at the front and extends at the rear. No matter how many consumers or producers wish to communicate, the buffer accommodates at most one of each on each full step of the successor function.

Process control Our final example of Exchange Functions is a process control program. We imagine that during some manufacturing process it is necessary to maintain a certain temperature distribution in a vat of liquid over a long period of time. The vat contains several controllers, several sensors (thermometers), and several heating elements at fixed locations. Each controller communicates directly, over its own specialized channels, with its own thermometer and its own heater. The controllers communicate with each other over a single, common channel.

Each controller's position is indicated by its $\langle i, j\rangle$ coordinates (Figure 7-1). The controller's state is a finite buffer in which it maintains the last few readings it has received. A reading is an ordered pair of the form ⟨position, temperature⟩. By the analysis of these values, the controller decides which instructions to send to its heater. The controller's program is a five-state loop. (1) It gets a reading from its own thermometer. (2) It offers its ⟨position, reading⟩ pair on the common controller channel, world. (3) When it receives a ⟨position, reading⟩ pair from some other controller, it adds it to its finite buffer. (4) It analyzes the updated data and decides what instructions to send to its heater. (5) Finally, it deletes its oldest datum, leaving room for a new reading on the next step.

The heating element repeatedly receives instructions and adjusts its control to follow those instructions. Each sensor (like the real-time clock) continuously offers the temperature to its controller. We use function proj-2, that evaluates both of its arguments and returns the second. This example uses exchange function X in a many-to-many organization.

Figure 7-1 The process control bath.

$$
\begin{aligned}
&\text{successor (controller[i,j])} \equiv\\
&\qquad \text{process[i,j] (last-insert (controller,}\\
&\qquad\qquad\qquad\qquad\qquad \mathsf{X}_{\text{world}}(\langle\langle \text{i, j}\rangle,\ \mathsf{X}_{\text{sensor[i,j]}}\ \text{(ack)}\rangle))).
\end{aligned}
$$

$$
\begin{aligned}
&\text{process[i,j] (controller)} \equiv\\
&\qquad \text{proj-2 (}\langle \mathsf{X}_{\text{heater[i,j]}}\ \text{(decide-what-to-send-to-heater (controller)),}\\
&\qquad\qquad \text{rest (controller)}\rangle).
\end{aligned}
$$

$$
\begin{aligned}
&\text{successor (sensor[i,j])} \equiv\\
&\qquad \mathsf{XR}_{\text{sensor[i,j]}}\ \text{(current-temperature-register[i,j])}.
\end{aligned}
$$

$$
\begin{aligned}
&\text{successor (heater[i,j])} \equiv\\
&\qquad \text{adjust-control[i,j] (}\mathsf{X}_{\text{heater[i,j]}}\text{(ack))}.
\end{aligned}
$$

Guarded Exchange Functions

These next two sections describe some possible extensions to the Exchange Functions model.

Many systems use a variant of Dijkstra's guarded commands (Section 2-2) to combine indeterminacy and communication selection. The original exchange functions definition has no mechanism for requesting an exchange on one of several channels. Let us consider the effect of extending exchange functions to include a primitive with the power of guarded commands.

In 1963, John McCarthy introduced the operator amb for effecting nondeterminism [McCarthy 63]. McCarthy's amb is a binary operator. Its value is whichever of its two operands is defined. If both are defined, then amb can return either one; if neither is defined, then amb is also undefined. Operationally, amb (f, g) can be thought of as "start both f and g, returning whichever finishes 'first.'" Of course, undefined operands never finish. Since these systems are formally time-free, a complex computation may finish before a simple one. For this section, we restrict the operands of amb to be calls to exchange functions. For any two exchange calls, E_α and E_β, we let amb (E_α, E_β) be whichever of the two exchanges matches first. If there are waiting exchanges on both channels, then we let amb choose indeterminately which one to match. We allow amb to range over any variety of exchange function. Additionally, we extend amb to take an arbitrary number of arguments. Thus, a typical subexpression of a successor function using our amb is:

$$\mathsf{amb}\ (\mathsf{E}_\alpha(1),\ \mathsf{E}_\beta(2),\ \mathsf{E}_\gamma(3),\ \mathsf{E}_\delta(4))$$

The evaluation of this expression sends communication offers out along channels α, β, γ, and δ. When one is accepted, the other three offers are rescinded and values are exchanged on the successful channel.

One peculiarity of this naive introduction of amb is that unlike guarded commands, the calling function cannot find out which amb branch was selected. All that is returned is the resulting value. This difficulty can be overcome by having each sending process decorate its message with the identity of its communication channel. A second artifact is that since XR exchanges always return immediately, an XR in an amb may dominate the other arguments. This implies that XRs inside ambs are of limited utility.

Delaying Exchange

Calling an exchange function produces communication exactly when the function returns. Let us assume that process A wishes to exchange on channel α. If no process is waiting to exchange on α, A either waits for a match (if it executed X or XM) or immediately returns, reporting failure (if it did an XR).

One could imagine other possible timing arrangements for exchanges. A process might expect to have a use for an exchange value, but have another useful computation to do in the meantime. Doing this other computation might reveal that the information requested in the original exchange was not really needed after all. We consider the possibility of allowing the process to initiate an exchange and continue with its computation, pausing only when the value is really needed. This variation is inspired by the theme of *call-by-need*: delaying parameter evaluation until use. This pattern can also be viewed as treating message communication as a fork operation, where using the returned value is the occasion to do the join. We call this mode of communication *join-by-need*.

Starting an exchange and completing it later has different meanings for the waiting (X, XM) and instantaneous (XR) functions. For the waiting exchange primitives, the desired implementation is a simple fork and join. When the interpreter sees a call to an X or XM exchange function, it initiates the exchange.

The process is free to continue its processing. Only when the value returned by the exchange function is used in the computation is there a potential for delay. If the exchange has not been completed by that time, the process waits for the value.

Sometimes the value of the exchange function is never used. For example, acknowledgments (such as the acknowledgment in the buffer insertion example) are frequently not examined. Here the forking acts like a send-and-forget message-passing system. Even if the value is not examined, the exchange is still deemed to have taken place.

The semantics of programs incorporating this fork-and-join primitive differ from those of the original waiting system. Most significantly, exchange no longer effects synchronization. For example, an unchecked semaphore no longer synchronizes. To restore the synchronization aspect of the exchange, we would need an "exchange and immediately access primitive."

The instantaneous exchange function (XR) presents an opportunity to introduce a new primitive. In the original call-by-value semantics, the XR exchange function implied an "instantaneous" exchange. If one considers the intent of XR as an exchange without waiting, the delayed evaluation metaphor can provide a different meaning. A call to XR signifies an exchange offer that can later be withdrawn. If the offer is not accepted before the answer is needed, then the usual failure response (return of the original argument) is given. The notion of a time-out is a traditional one in operating systems theory. Our new primitive allows a "compute out." If further computation reveals that the value would not be useful unless it were immediately present, then the exchange is aborted.

The following analogy may prove helpful. Imagine (process) Joe in his office. He is researching some problem (computing) and decides he needs to go to the library (obtain some resource). Joe can take a cab or a bus to the library (Joe has the choice of two different ways of obtaining the resource). Cabs are preferable, but the taxi company is unreliable and sometimes does not respond to requests. Joe decides that he might want to take a cab to the library later, so he sends out an XR_{cab}(request for cab) and continues his research (computes). At some point he may decide that he really does not need to go to the library (the computation never needs the value in the library). He can then just forget to see if any cab ever responded to his request. The cab may or may not eventually appear.* Or Joe could find himself stuck, with no choice but to go to the library. If no cab appears (the XR responded with his original argument), he can give up and take the bus.

The delayed exchange system can be integrated with the amb operator of the previous section. Joe could then call a cab, call a bus, and compute. When his computation became limited by the need to visit the library, he could then wait for either the cab or the bus to arrive.

* This attitude on the part of their customers may explain why the cabs are so unreliable.

As powerful as this extension may seem, it has some limitations. For example, a process cannot discover if an exchange has been completed without either forcing the exchange to complete or abort. This parallels the inability of a function in a delayed-evaluation system (Chapter 12) to find out if a value has been obtained for a delayed evaluation.

Since the XR operator possesses duration in this scheme, we can imagine another exchange function that would fit into the previously barred portion of Table 7-2. This function would be both instantaneous (nonblocking) and able to converse with itself.

Perspective

Exchange Functions provides bidirectional, synchronous communication. The model includes mechanisms for blocking communication, unblocked communication, and broadcast offers of communication. Exchange Functions also extends neatly to other capabilities, such as guarded commands and call-by-need. Syntactically, Exchange Functions has a particularly simple and elegant form: a minimal amount of structure provides a general and flexible facility.

This is not to imply that Exchange Functions handles all synchronization problems. In particular, the static number of exchange channels and the inability to evaluate the exchange channel before use are liabilities for the description of dynamically growing systems. However, these deficiencies are easily remedied. Overall, Exchange Functions allows many interesting communication architectures to be built from only a few simple primitives.

PROBLEMS

7-1 Program a general (n-ary) semaphore in Exchange Functions. Base your program on the binary semaphore program.

7-2 Rewrite the buffer program to be a bounded buffer.

7-3 Rewrite the problem of the constant-temperature liquid bath so that it is more realistic.

† **7-4** A non-empty channel can be in one of two states: either there is a single waiting X or there are one or more waiting XMs. Use this information to design a bounded-time program and data structure to perform the channel operations.

7-5 Exchange Functions hypothesizes three classes of exchange functions with communication capabilities represented by Table 7-1. Invent new classes of exchange functions, describing their possible communication patterns. Present a rationale for your system.

7-6 What is the effect of allowing amb to range over any expression?

REFERENCES

[**Fitzwater 77**] Fitzwater, D. R., and P. Zave, "The Use of Formal Asynchronous Process Specifications in a System Development Process," *Proc. 6th Texas Conf. Comp. Syst.*, The University of Texas at Austin (November 1977), pp. 2B-21:2B-30. This paper contains the first published reference to exchange functions. At that time, they were called XC, XA, and XS (now X, XM, and XR).

[**McCarthy 63**] McCarthy, J., "A Basis for a Mathematical Theory of Computation," in P. Braffort, and D. Hirschberg (eds.), *Computer Programming and Formal Systems*, North Holland, Amsterdam (1963), pp. 33–70. In this paper McCarthy lays the groundwork for many of the ideas that have grown up around the mathematical theory of computation, such as the idea that programs can be proven correct. He also introduces the amb primitive, an idea much copied and insufficiently credited.

[**Metcalfe 76**] Metcalfe, R. M., and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *CACM*, vol. 19, no. 7 (July 1976), pp. 395–404. A description of the Ethernet mechanism for local-area networks. Ethernet uses a contention ring, where processors communicate by "yelling" into a shared medium (coaxial cable) until one can be heard by itself.

[**Zave 77**] Zave, P., and D. R. Fitzwater, "Specification of Asynchronous Interactions using Primitive Functions," Technical Report 598, Department of Computer Science, University of Maryland, College Park, Maryland (1977). This paper is a good description of the Exchange Functions model. We adapted the unbounded buffer example from this paper.

[**Zave 82**] Zave, P., "An Operational Approach to Requirements Specification for Embedded Systems," *IEEE Trans. Softw. Eng.*, vol. SE-8, no. 3 (May 1982), pp. 250–269. This paper presents Exchange Functions as a requirements specification mechanism. Zave emphasizes using Exchange Functions as an operational approach to requirements specification. Operational approaches produce executable descriptions of the system specified. (This is in contrast to declarative approaches that specify properties of the system without describing the mechanism for achieving them.) Zave also describes the specification language PAISLey, which is based on exchange functions.

[**Zave 83**] Zave, P., personal communication, 1983.